Figures

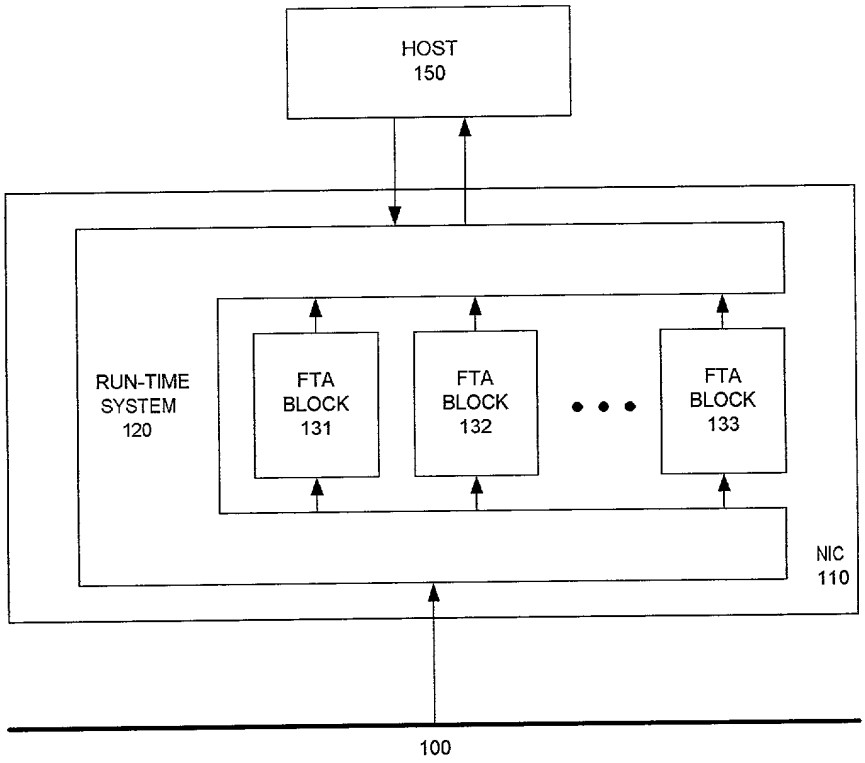
HOST
150
RUN-TIME
SYSTEM
120
FTA
BLOCK
131
FTA
BLOCK
132
FTA
BLOCK
133
NIC
110
100

FIG. 1

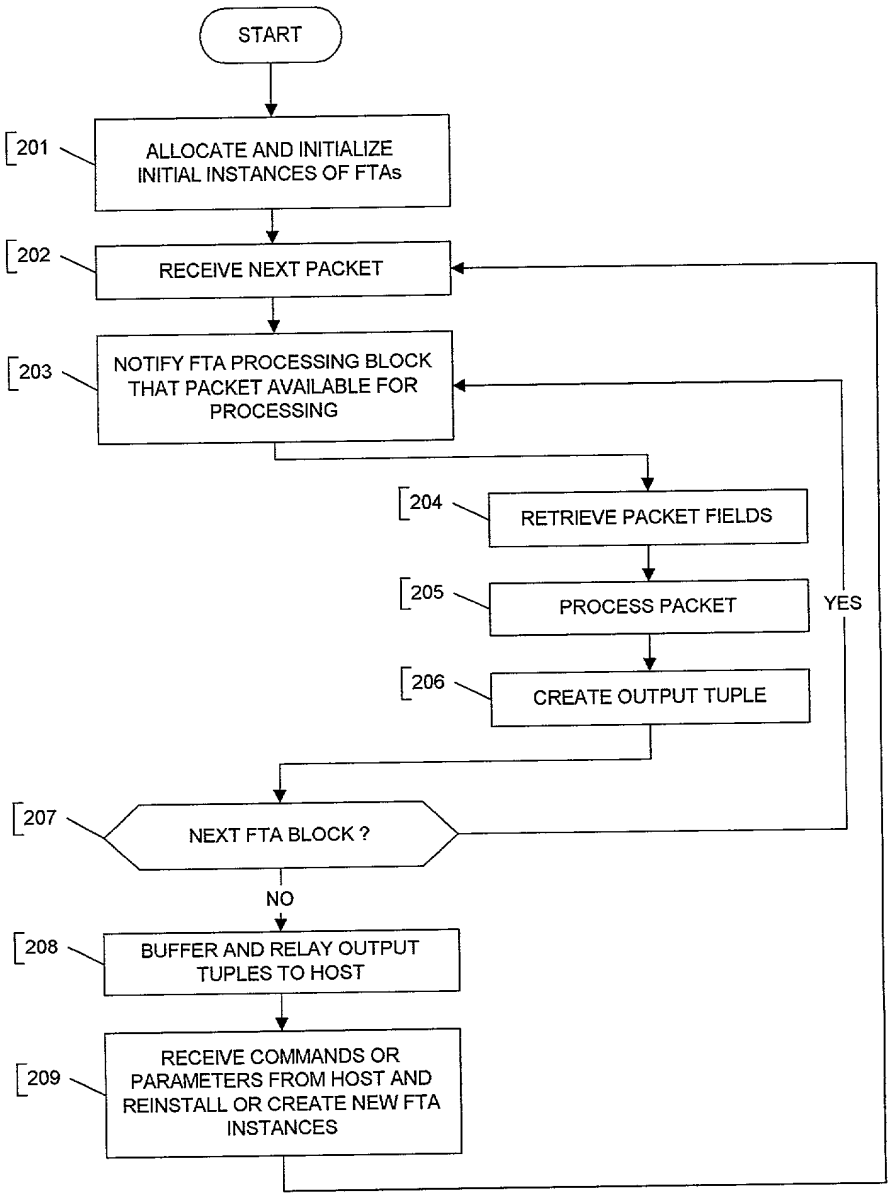
START
201
ALLOCATE AND INITIALIZE INITIAL INSTANCES OF FTAs
202
RECEIVE NEXT PACKET
203
NOTIFY FTA PROCESSING BLOCK THAT PACKET AVAILABLE FOR PROCESSING
204
RETRIEVE PACKET FIELDS
205
PROCESS PACKET
206
CREATE OUTPUT TUPLE
207
NEXT FTA BLOCK ?
YES
NO
208
BUFFER AND RELAY OUTPUT TUPLES TO HOST
209
RECEIVE COMMANDS OR PARAMETERS FROM HOST AND REINSTALL OR CREATE NEW FTA INSTANCES

FIG. 2

```
301 DEFINE{
302 fta_name 'count_pkts';
303 }
304
305 select timestamp, hdr_length
306 from IPV4 p
307 where hdr_length > 50
```

FIG. 3

```
601 DEFINE{
602 fta_name 'count_pkts';
603 aggregate_slots '1';
604 }
605
606 select timebucket, count(*)
607 from IPV4 p
608 group by timestamp/5000 AS timebucket
```

FIG. 6

```
901 DEFINE{
902 fta_name 'count_pkts';
903 }
904
905 select timestamp, hdr_length, count(*),
906       sum(offset), max(ttl), min(destIP)
907 from IPV4 p
908 where ttl in [ 2, 3, 6, 9 ] and
909       timestamp > (TIMEVAL '123.45') + 5
910 group by timestamp, hdr_length
911
```

FIG. 9

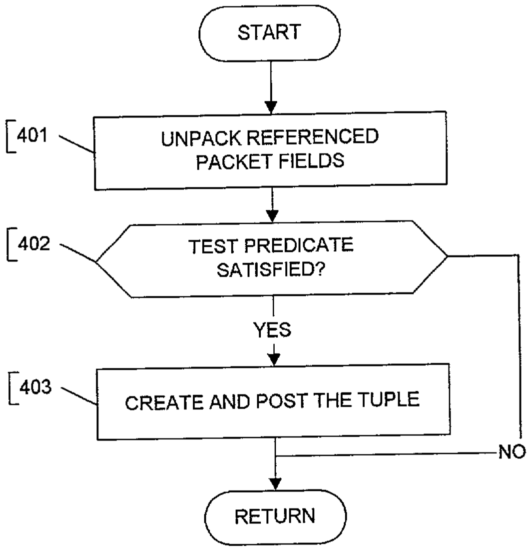

FIG. 4

```
501 #include "rts.h"
502 #include "fta.h"
503
504
505
506 /*              The FTA references the following internal fcns:
507 */
508
509 struct count_pkts_fta{
510         struct FTA f;
511 };
512
513 struct count_pkts_tuple{
514         struct timeval  tuple_var0;
515         unsigned int  tuple_var1;
516 };
517
518 static int free_fta(struct FTA *f){
519         return 0;
520 }
521
522 static int control_fta(struct FTA *f,  int command, int sz, void *value){
523         struct count_pkts_fta * t = (struct count_pkts_fta *) f;
524
525         return 0;
526 }
527
528 static int accept_packet(struct FTA *f, struct packet *p){
529 /*              Variables which are always needed     */
530         int retval, tuple_size, tuple_pos;
531         struct count_pkts_tuple *tuple;
532         struct count_pkts_fta *t = (struct count_pkts_fta*) f;
533
534 /*              Variables for unpacking attributes    */
535         unsigned int  unpack_var_hdr_length_3;
536         struct timeval  unpack_var_timestamp_3;
537
538
539 /*              Unpack the referenced fields  */
540         retval =  get_ipv4_hdr_length(p, &unpack_var_hdr_length_3);
541         if(retval) return 0;
542         retval =  get_timestamp(p, &unpack_var_timestamp_3);
543         if(retval) return 0;
544
545
546 /*              Test the predicate     */
547         if( !( ( unpack_var_hdr_length_3>50 ) ) )
548                 return 0;
549
550 /*              Create and post the tuple     */
551         tuple_size = sizeof( struct count_pkts_tuple);
552         tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
553         if( tuple == NULL)
554                 return 0;
555         tuple_pos = sizeof( struct count_pkts_tuple);
556         tuple->tuple_var0 = unpack_var_timestamp_3;
557         tuple->tuple_var1 = unpack_var_hdr_length_3;
558         post_tuple(tuple);
559
560         return 0;
561 }
```

FIG. 5A

```
562 struct FTA * count_pkts_fta_alloc(unsigned stream_id, unsigned priority, int
563 argvc, void * argv[]){
564         struct count_pkts_fta* f;
565
566         if((f=fta_alloc(0,sizeof(struct count_pkts_fta)))==0){
567                 return(0);
568         }
569
570         f->f.stream_id=stream_id;
571         f->f.priority=priority;
572         f->f.alloc_fta=count_pkts_fta_alloc;
573         f->f.free_fta=free_fta;
574         f->f.control_fta=control_fta;
575         f->f.accept_packet=accept_packet;
576
577         return (struct FTA *) f;
578 }
```

FIG. 5B

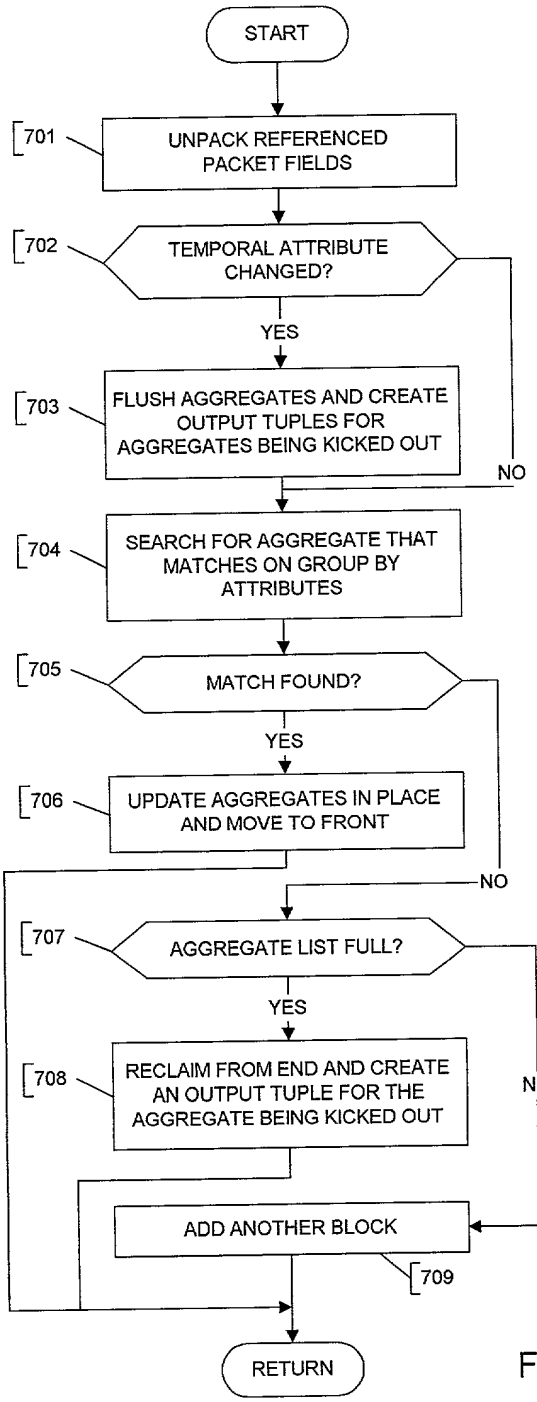
START
701
UNPACK REFERENCED PACKET FIELDS
702
TEMPORAL ATTRIBUTE CHANGED?
YES
NO
703
FLUSH AGGREGATES AND CREATE OUTPUT TUPLES FOR AGGREGATES BEING KICKED OUT
704
SEARCH FOR AGGREGATE THAT MATCHES ON GROUP BY ATTRIBUTES
705
MATCH FOUND?
YES
NO
706
UPDATE AGGREGATES IN PLACE AND MOVE TO FRONT
707
AGGREGATE LIST FULL?
YES
NO
708
RECLAIM FROM END AND CREATE AN OUTPUT TUPLE FOR THE AGGREGATE BEING KICKED OUT
ADD ANOTHER BLOCK
709
RETURN

FIG. 7

```
#include "rts.h"
#include "fta.h"


/*              The FTA references the following internal fcns:
        Divide_Timeval_Int
*/

static struct timeval Divide_Timeval_Int(struct timeval t, int d){
        struct timeval r;
        r.tv_sec = t.tv_sec / d;
        r.tv_usec = (t.tv_usec + 1000*( t.tv_sec % d )) / d;
        return(r);
}


struct count_pkts_aggr_struct{
        struct timeval  gb_var0;
        unsigned int  aggr_var0;
        struct count_pkts_aggr_struct *next;
};

struct count_pkts_fta{
        struct FTA f;
        struct count_pkts_aggr_struct *aggr_head;
        int n_aggrs;
        int max_aggrs;
        struct timeval  last_gb_0;
};

struct count_pkts_tuple{
        struct timeval  tuple_var0;
        unsigned int  tuple_var1;
};

static void fta_aggr_flush(struct FTA *f){
        struct count_pkts_aggr_struct *curr_aggr, *next_aggr;
        int tuple_size;
        struct count_pkts_tuple *tuple;
        struct count_pkts_fta * t = (struct count_pkts_fta *) f;

        curr_aggr = t->aggr_head;
        while(curr_aggr != NULL){
                next_aggr = curr_aggr->next;
/*              Create an output tuple for the aggregate being kicked out   */
                        tuple_size = sizeof( struct count_pkts_tuple);
                        tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
                        if( tuple != NULL){
                                tuple_pos = sizeof( struct count_pkts_tuple);
                                tuple->tuple_var0 = curr_aggr->gb_var0;
                                tuple->tuple_var1 = curr_aggr->aggr_var0;
                                post_tuple(tuple);
                        }
                fta_free(f,curr_aggr);
                curr_aggr = next_aggr;
        }
        t->n_aggrs = 0;
        t->aggr_head = NULL;
}
```

FIG. 8A

```
801 static int free_fta(struct FTA *f){
802         fta_aggr_flush();
803         return 0;
804 }
805
806 static int control_fta(struct FTA *f,  int command, int sz, void *value){
807         struct count_pkts_fta * t = (struct count_pkts_fta *) f;
808
809         if(command == FTA_COMMAND_FLUSH)
810                 fta_aggr_flush();
811         return 0;
812 }
813
814 static int accept_packet(struct FTA *f, struct packet *p){
815 /*              Variables which are always needed     */
816         int retval, tuple_size, tuple_pos;
817         struct count_pkts_tuple *tuple;
818         struct count_pkts_fta *t = (struct count_pkts_fta*) f;
819
820 /*              Variables for unpacking attributes    */
821         struct timeval  unpack_var_timestamp_3;
822
823
824
825 /*              Variables for aggregation     */
826 /*              Group-by attributes    */
827         struct timeval  gb_attr_0;
828
829 /*              Variables for manipulating the aggregate list        */
830         struct count_pkts_aggr_struct *curr_aggr, *prev_aggr;
831
832 /*              Unpack the referenced fields  */
833         retval =  get_timestamp(p, &unpack_var_timestamp_3);
834         if(retval) return 0;
835
836
837 /*               (no predicate to test)       */
838
839 /*              Search for an aggregate that matches on the group by attributes     */
840         gb_attr_0 = Divide_Timeval_Int(unpack_var_timestamp_3, 5000);
841
842 /*              Flush the aggregates if the temporal gb attrs have changed. */
843         if( !( (Compare_Timeval(t->last_gb_0, gb_attr_0) == 0) ) )
844                 fta_aggr_flush();
845
846         curr_aggr = t->aggr_head; prev_aggr = NULL;
847         while(curr_aggr != NULL){
848                 if( (Compare_Timeval(gb_attr_0, curr_aggr->gb_var0) == 0) )
849                         break;
850                 if(curr_aggr->next != NULL)
851                         prev_aggr = curr_aggr;
852                 curr_aggr = curr_aggr->next;
853         }
854
```

FIG. 8B

```
801         if(curr_aggr != NULL){
802 /*              Match found : update in place, move to front.       */
803                 curr_aggr->aggr_var0++;
804
805                 if(prev_aggr != NULL)
806                         prev_aggr->next = curr_aggr->next;
807                 if(t->aggr_head != curr_aggr)
808                         curr_aggr->next = t->aggr_head;
809                 t->aggr_head = curr_aggr;
810         }else{
811 /*              No match found ...     */
812                 if(t->n_aggrs == t->max_aggrs){
813 /*              And the aggregate list is full.  Reclaim from the end.      */
814                         if(prev_aggr != NULL)
815                                 curr_aggr = prev_aggr->next;
816                         else     curr_aggr = t->aggr_head;
817                         if(prev_aggr != NULL)
818                                 prev_aggr->next = curr_aggr->next;
819                         if(t->aggr_head != curr_aggr) curr_aggr->next = t->aggr_head;
820                         t->aggr_head = curr_aggr;
821
822
823 /*              Create an output tuple for the aggregate being kicked out   */
824                         tuple_size = sizeof( struct count_pkts_tuple);
825                         tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
826                         if( tuple != NULL){
827                                 tuple_pos = sizeof( struct count_pkts_tuple);
828                                 tuple->tuple_var0 = curr_aggr->gb_var0;
829                                 tuple->tuple_var1 = curr_aggr->aggr_var0;
830                                 post_tuple(tuple);
831                         }
832                 }else{
833 /*              Room in the aggregate list, add another block.      */
834                         curr_aggr = (struct count_pkts_aggr_struct *)
835 fta_alloc(f,sizeof(struct count_pkts_aggr_struct) );
836                         if(curr_aggr == NULL) return(0);
837                         curr_aggr->next = t->aggr_head;
838                         t->aggr_head = curr_aggr;
839                         t->n_aggrs++;
840                 }
841
842                 curr_aggr->gb_var0 = gb_attr_0;
843                 curr_aggr->aggr_var0 = 1;
844         }
845
846         return 0;
847 }
```

FIG. 8C

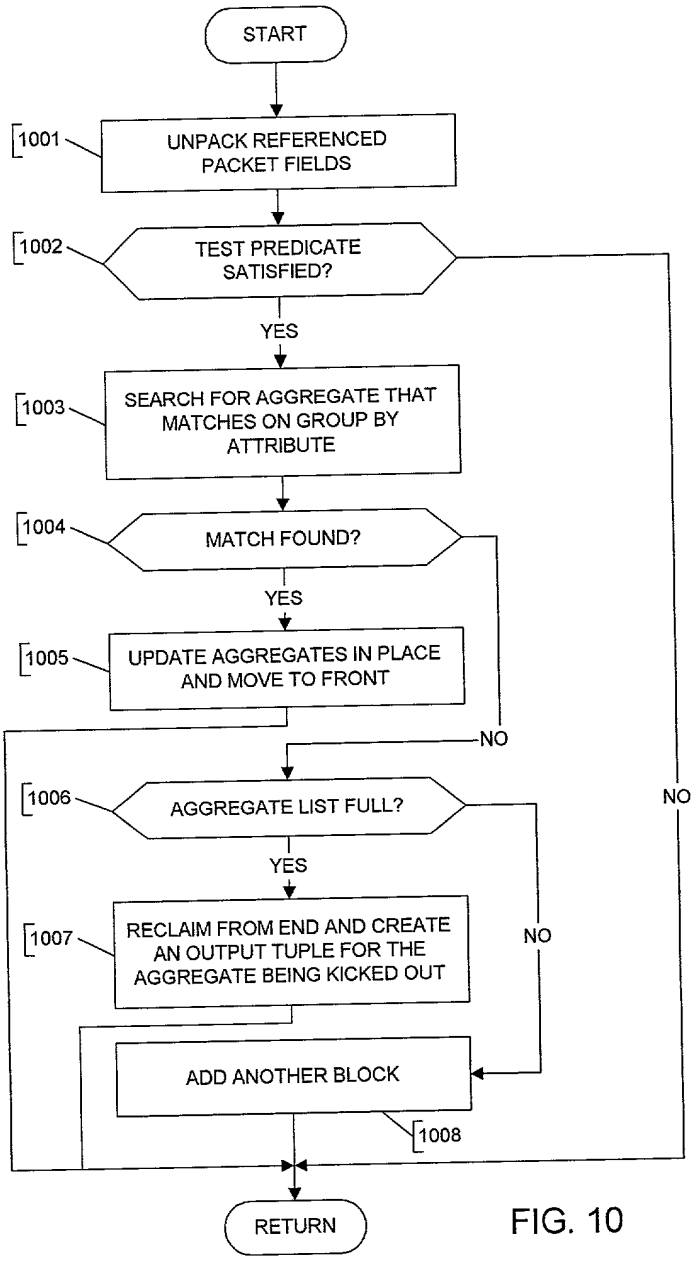
START
1001
UNPACK REFERENCED PACKET FIELDS
1002
TEST PREDICATE SATISFIED?
YES
NO
1003
SEARCH FOR AGGREGATE THAT MATCHES ON GROUP BY ATTRIBUTE
1004
MATCH FOUND?
YES
NO
1005
UPDATE AGGREGATES IN PLACE AND MOVE TO FRONT
1006
AGGREGATE LIST FULL?
YES
NO
1007
RECLAIM FROM END AND CREATE AN OUTPUT TUPLE FOR THE AGGREGATE BEING KICKED OUT
ADD ANOTHER BLOCK
1008
RETURN

FIG. 10

```
1101 #include "rts.h"
1102 #include "fta.h"
1103
1104
1105
1106 /*              The FTA references the following internal fcns:
1107         Add_Timeval_Int
1108         Compare_Timeval
1109         Subtract_Timeval_Timeval
1110         Timeval_Constructor
1111 */
1112
1113 static struct timeval Add_Timeval_Int(struct timeval t, int inc){
1114         struct timeval r;
1115         r.tv_usec = t.tv_usec + (inc % 1000);
1116         r.tv_sec = t.tv_sec + inc / 1000;
1117         if(r.tv_usec > 999){
1118                 r.tv_usec -= 1000;
1119                 r.tv_sec++;
1120         }
1121 }
1122
1123 static int Compare_Timeval(struct timeval t1, struct timeval t2){
1124         return( t1.tv_sec != t2.tv_sec ? t1.tv_sec - t2.tv_sec : t1.tv_usec -
1125 t2.tv_usec );
1126 }
1127
1128 static int Subtract_Timeval_Timeval(struct timeval t1, struct timeval t2){
1129         return(1000*(t1.tv_sec - t2.tv_sec) + (t1.tv_usec - t2.tv_usec) );
1130 }
1131
1132 static struct timeval Timeval_Constructor(int s, int m){
1133         struct timeval r;
1134         r.tv_sec = s;
1135         r.tv_usec = m;
1136         return(r);
1137 }
1138
1139 struct count_pkts_aggr_struct{
1140         struct timeval  gb_var0;
1141         unsigned int  gb_var1;
1142         unsigned int  aggr_var0;
1143         unsigned int  aggr_var1;
1144         unsigned int  aggr_var2;
1145         unsigned int  aggr_var3;
1146 },
1147
1148 struct count_pkts_fta{
1149         struct FTA f;
1150         struct count_pkts_aggr_struct *aggr_head;
1151         int n_aggrs;
1152         int max_aggrs;
1153 };
1154
1155 struct count_pkts_tuple{
1156         struct timeval  tuple_var0;
1157         unsigned int  tuple_var1;
1158         unsigned int  tuple_var2;
1159         unsigned int  tuple_var3;
1160         unsigned int  tuple_var4;
1161         unsigned int  tuple_var5;
1162 };
```

FIG. 11A

```
1101 static int free_fta(struct FTA *f){
1102         struct count_pkts_aggr_struct *curr_nd, *next_nd;
1103         curr_nd = f->aggr_head;
1104         while(curr_nd != NULL){
1105                 next_nd = curr_nd->next;
1106                 fta_free(f, curr_nd);
1107                 curr_nd = next_nd;
1108         }
1109         return 0;
1110 }
1111
1112 static int control_fta(struct FTA *f,  int command, int sz, void *value){
1113         struct count_pkts_fta * t = (struct count_pkts_fta *) f;
1114         return 0;
1115 }
1116
1117 static int accept_packet(struct FTA *f, struct packet *p){
1118 /*              Variables which are always needed     */
1119         int retval, tuple_size, tuple_pos;
1120         struct count_pkts_tuple *tuple;
1121         struct count_pkts_fta *t = (struct count_pkts_fta*) f;
1122
1123 /*              Variables for unpacking attributes    */
1124         unsigned int    unpack_var_destIP_3;
1125         unsigned int    unpack_var_hdr_length_3;
1126         unsigned int    unpack_var_offset_3;
1127         struct timeval  unpack_var_timestamp_3;
1128         unsigned int    unpack_var_ttl_3;
1129
1130 /*              Variables for aggregation     */
1131 /*              Group-by attributes    */
1132         struct timeval  gb_attr_0;
1133         unsigned int  gb_attr_1;
1134
1135 /*              Variables for manipulating the aggregate list        */
1136         struct count_pkts_aggr_struct *curr_aggr, *prev_aggr;
1137
1138 /*              Unpack the referenced fields  */
1139         retval = get_ipv4_dest_ip(p, &unpack_var_destIP_3);
1140         if(retval) return 0;
1141         retval = get_ipv4_hdr_length(p, &unpack_var_hdr_length_3);
1142         if(retval) return 0;
1143         retval = get_ipv4_offset(p, &unpack_var_offset_3);
1144         if(retval) return 0;
1145         retval = get_timestamp(p, &unpack_var_timestamp_3);
1146         if(retval) return 0;
1147         retval = get_ipv4_ttl(p, &unpack_var_ttl_3);
1148         if(retval) return 0;
1149
1150 /*               Test predicate                */
1151         if( !( ( ( ( ( unpack_var_ttl_3 == 2 ) || ( unpack_var_ttl_3 == 3 ) ||
1152 ( unpack_var_ttl_3 == 6 ) || ( upack_var_ttl_3 == 9 ) ) ) &&
1153 ( Compare_Timeval(unpack_var_timestamp_3, Add_Timeval_Int(Timeval_Constructor(123,
1154 450), 5)) >0 ) ) ) )
1155         return 0;
1156
```

FIG. 11B

```
1101 /*             Search for an aggregate that matches on the group by attributes    */
1102         gb_attr_0 = unpack_var_timestamp_3;
1103         gb_attr_1 = unpack_var_hdr_length_3;
1104         curr_aggr = t->aggr_head; prev_aggr = NULL;
1105         while(curr_aggr != NULL){
1106                 if( (Compare_Timeval(gb_attr_0, curr_aggr->gb_var0) == 0) &&
1107 (gb_attr_1 == curr_aggr->gb_var1 ) )
1108                         break;
1109                 if(curr_aggr->next != NULL)
1110                         prev_aggr = curr_aggr;
1111                 curr_aggr = curr_aggr->next;
1112         }
1113
1114         if(curr_aggr != NULL){
1115 /*              Match found : update in place, move to front.       */
1116                 curr_aggr->aggr_var0++;
1117                 curr_aggr->aggr_var1 += unpack_var_offset_3;
1118                 curr_aggr->aggr_var2 = ( curr_agg->aggr_var2 >= unpack_var_ttl_3 ?
1119 curr_aggr->aggr_var2 : unpack_var_ttl_3 );
1120                 curr_aggr->aggr_var3 = ( curr_agg->aggr_var3 <=
1121 unpack_var_destIP_3 ? curr_aggr->aggr_var3 : unpack_var_destIP_3 );
1122                 if(prev_aggr != NULL)
1123                         prev_aggr->next = curr_aggr->next;
1124                 if(t->aggr_head != curr_aggr)
1125                         curr_aggr->next = t->aggr_head;
1126                 t->aggr_head = curr_aggr;
1127         }else{
1128 /*              No match found ...     */
1129                 if(t->n_aggrs == t->max_aggrs){
1130 /*              And the aggregate list is full.  Reclaim from the end.      */
1131                         if(prev_aggr != NULL)
1132                                 curr_aggr = prev_aggr->next;
1133                         else     curr_aggr = t->aggr_head;
1134                         if(prev_aggr != NULL)
1135                                 prev_aggr->next = curr_aggr->next;
1136                         if(t->aggr_head != curr_aggr) curr_aggr->next =
1137 t->aggr_head;
1138                         t->aggr_head = curr_aggr;
1139
1140 /*              Create an output tuple for the aggregate being kicked out   */
1141                         tuple_size = sizeof( struct count_pkts_tuple);
1142                         tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
1143                         if( tuple != NULL){
1144                                 tuple_pos = sizeof( struct count_pkts_tuple);
1145                                 tuple->tuple_var0 = curr_aggr->gb_var0;
1146                                 tuple->tuple_var1 = curr_aggr->gb_var1;
1147                                 tuple->tuple_var2 = curr_aggr->aggr_var0;
1148                                 tuple->tuple_var3 = curr_aggr->aggr_var1;
1149                                 tuple->tuple_var4 = curr_aggr->aggr_var2;
1150                                 tuple->tuple_var5 = curr_aggr->aggr_var3;
1151                                 post_tuple(tuple);
1152                         }
1153                 }else{
1154 /*              Room in the aggregate list, add another block.      */
1155                         curr_aggr = (struct count_pkts_aggr_struct *)
1156 fta_alloc(f,sizeof(struct count_pkts_aggr_struct) );
1157                         if(curr_aggr == NULL) return(0);
1158                         curr_aggr->next = t->aggr_head;
1159                         t->aggr_head = curr_aggr;
1160                         t->n_aggrs--;
1161                 }
1162
1163                 curr_aggr->gb_var0 = gb_attr_0;
1164                 curr_aggr->gb_var1 = gb_attr_1;
1165                 curr_aggr->aggr_var0 = 1;
1166                 curr_aggr->aggr_var1 = unpack_var_offset_3;
1167                 curr_aggr->aggr_var2 = unpack_var_ttl_3;
1168                 curr_aggr->aggr_var3 = unpack_var_destIP_3;
1169         }
1170
1171         return 0;
1172 }
```

FIG. 11C

```
1101 struct FTA * count_pkts_fta_alloc(unsigned stream_id, unsigned priority, int
1102 argvc, void * argv[]){
1103        struct count_pkts_fta* f;
1104
1105        if((f=fta_alloc(0,sizeof(struct count_pkts_fta)))==0){
1106               return(0);
1107        }
1108        f->aggr_head = NULL;
1109        f->n_aggrs = 0;
1110        f->max_aggrs = 1;
1111
1112        f->f.stream_id=stream_id;
1113        f->f.priority=priority;
1114        f->f.alloc_fta=count_pkts_fta_alloc;
1115        f->f.free_fta=free_fta;
1116        f->f.control_fta=control_fta;
1117        f->f.accept_packet=accept_packet;
1118
1119        return (struct FTA *) f;
1120 }
```

FIG. 11D

```
1201 DEFINE{
1202 fta_name 'test_query';
1203 }
1204
1205 select hdr_length,  max( str_find_substr(IPv4_header, 'bob') ),
1206             str_find_substr( min(IPv4_header) , 'bob')
1207 from IPV4 p
1208 where precedence > 5 and IPv4_header >
1209             str_find_substr(IPv4_data, 'host:*\n')
1210 group by hdr_length
```

FIG. 12

```
1301 DEFINE{
1302 fta_name 'count_pkts';
1303 min_hdr_length 'int';
1304 }
1305
1306 select timestamp, hdr_length
1307 from IPV4 p
1308 where hdr_length > $min_hdr_length
```

FIG. 13

```
1401 #include "pts.h"
1402 #include "fta.h"
1403
1404
1405
1406 /*            The FTA references the following internal fcns:
1407 */
1408
1409 struct count_pkts_fta{
1410         struct FTA f;
1411         int  param_min_hdr_length;
1412 };
1413
1414 struct count_pkts_tuple{
1415         unsigned long long int  tuple_var0;
1416         unsigned int  tuple_var1;
1417 };
1418
1419 static void load_params(struct count_pkts_fta *t, int sz, void *value){
1420         int pos=0;
1421         int data_pos;
1422
1423         data_pos = sizeof( int  );
1424         if(data_pos > sz) return;
1425
1426         t->param_min_hdr_length =  *( (int  *)( (char *)value+pos) );
1427         pos += sizeof( int  );
1428 }
1429
1430 static int free_fta(struct FTA *f){
1431         return 0;
1432 }
1433
1434 static int control_fta(struct FTA *f,  int command, int sz, void *value){
1435         struct count_pkts_fta * t = (struct count_pkts_fta *) f;
1436
1437         if(command == FTA_COMMAND_LOAD_PARAMS){
1438                 load_params(t, sz, value);
1439         }
1440         return 0;
1441 }
1442
1443 static int accept_packet(struct FTA *f, struct packet *p){
1444 /*            Variables which are always needed     */
1445         int retval, tuple_size, tuple_pos;
1446         struct count_pkts_tuple *tuple;
1447         struct count_pkts_fta *t = (struct count_pkts_fta*) f;
1448
1449 /*            Variables for unpacking attributes    */
1450         unsigned int  unpack_var_hdr_length_3;
1451         unsigned long long int  unpack_var_timestamp_3;
1452
1453
1454 /*            Unpack the referenced fields  */
1455         retval =  get_ipv4_hdr_length(p, &unpack_var_hdr_length_3);
1456         if(retval) return 0;
1457         retval =  get_timestamp(p, &unpack_var_timestamp_3);
1458         if(retval) return 0;
1459
```

FIG. 14A

```
1401  /*              Test the predicate      */
1402          if( !( ( unpack_var_hdr_length_3>t->param_min_hdr_length ) ) )
1403                  return 0;
1404
1405  /*              Create and post the tuple       */
1406          tuple_size = sizeof( struct count_pkts_tuple);
1407          tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
1408          if( tuple == NULL)
1409                  return 0;
1410          tuple_pos = sizeof( struct count_pkts_tuple);
1411          tuple->tuple_var0 = unpack_var_timestamp_3;
1412          tuple->tuple_var1 = unpack_var_hdr_length_3;
1413          post_tuple(tuple);
1414
1415          return 0;
1416  }
1417
1418  struct FTA * count_pkts_fta_alloc(unsigned stream_id, unsigned priority, int
1419  command,  int sz, void *value){
1420          struct count_pkts_fta* f;
1421
1422          if((f=fta_alloc(0,sizeof(struct count_pkts_fta)))==0)
1423                  return(0);
1424          }
1425
1426          f->f.stream_id=stream_id;
1427          f->f.priority=priority;
1428          f->f.alloc_fta=count_pkts_fta_alloc;
1429          f->f.free_fta=free_fta;
1430          f->f.control_fta=control_fta;
1431          f->f.accept_packet=accept_packet;
1432
1433          load_params(f, sz, value);
1434
1435          return (struct FTA *) f;
1436  }
```

FIG. 14B